



## King's Research Portal

### *Document Version*

Publisher's PDF, also known as Version of record

[Link to publication record in King's Research Portal](#)

### *Citation for published version (APA):*

Lano, K., & Yassipour-Tehrani, S. (2016). Verified bidirectional transformations by construction. *CEUR Workshop Proceedings*, 1693, 28-37. <http://ceur-ws.org/Vol-1693/VoltPaper2.pdf>

### **Citing this paper**

Please note that where the full-text provided on King's Research Portal is the Author Accepted Manuscript or Post-Print version this may differ from the final Published version. If citing, it is advised that you check and use the publisher's definitive version for pagination, volume/issue, and date of publication details. And where the final published version is provided on the Research Portal, if citing you are again advised to check the publisher's website for any subsequent corrections.

### **General rights**

Copyright and moral rights for the publications made accessible in the Research Portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognize and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the Research Portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the Research Portal

### **Take down policy**

If you believe that this document breaches copyright please contact [librarypure@kcl.ac.uk](mailto:librarypure@kcl.ac.uk) providing details, and we will remove access to the work immediately and investigate your claim.

# Verified Bidirectional Transformations by Construction

K. Lano, S. Yassipour-Tehrani

Dept. of Informatics  
King's College London  
London, UK  
Email: kevin.lano@kcl.ac.uk,  
sobhan.yassipour\_tehrani@kcl.ac.uk

**Abstract.** Bidirectional transformations (bx) are of increasing significance in model-based engineering. Currently bx are defined using a number of specialised transformation languages. In this paper we show how standard UML elements such as use cases and OCL constraints can be used to define bx, thus taking advantage of the wide industrial and educational adoption and support for UML. We define patterns and techniques for specifying bx so that they are correct by construction.

## 1 Introduction

Bidirectional model transformations (bx) provide a means to maintain a consistency relation  $R$  between two models which may both change.

Bx are significant for a wide range of model based engineering scenarios such as round-trip engineering; automatic consistency maintenance of multiple inter-related models, and application and database co-evolution [3].

Bx are characterised by a binary relation  $R : SL \leftrightarrow TL$  between a source language (metamodel)  $SL$  and a target language  $TL$ .  $R(m, n)$  holds for a pair of models  $m$  of  $SL$  and  $n$  of  $TL$  when the models consist of data which corresponds under  $R$ . It should be possible to automatically derive from the definition of  $R$  both forward and reverse transformations  $R^\rightarrow : SL \times TL \rightarrow TL$  and  $R^\leftarrow : SL \times TL \rightarrow SL$  which aim to establish/maintain  $R$  between their first (respectively second) and their result target (respectively source) models, given both existing source and target models.

Bx should satisfy two key conditions [16, 12]:

1. *Correctness*: the forward and reverse transformations derived from a relation  $R$  do establish  $R$ :  $R(m, R^\rightarrow(m, n))$  and  $R(R^\leftarrow(m, n), n)$  for each  $m : SL, n : TL$ .
2. *Hippocraticness*: if source and target models already satisfy  $R$  then the forward and reverse transformations do not modify the models:  $R(m, n) \Rightarrow R^\rightarrow(m, n) = n$  and  $R(m, n) \Rightarrow R^\leftarrow(m, n) = m$  for each  $m : SL, n : TL$ .

Additionally, updates to a model by  $R^\rightarrow$  or  $R^\leftarrow$  should be the minimal necessary to restore  $R$ : this is termed the *principle of least change* [13].

Section 2 describes current bx research, Section 3 describes some bx patterns, and Section 4 describes our approach to bx specification in UML. Section 5 provides a proof that the bx properties for  $R^\rightarrow$  are satisfied by transformations defined using our approach. Section 6 gives an evaluation of the approach.

## 2 Bx Specification Languages and Techniques

Bx are the subject of extensive theoretical research [16, 17, 4, 12] and an annual conference, BX. The most well-known transformation language which supports definition of bx is QVT-R [14]. However, QVT-R has had limited uptake because of its complex semantics and specification style, compared to simpler unidirectional MT languages such as ATL [6] and ETL [7]. Triple Graph Grammars (TGG) are a graph transformation-based formalism which supports bx definition, although with a restricted expression language compared to OCL. Other approaches, such as JTL [5], and extensions of TGG [1], use constraint-based programming techniques to interpret relations  $P(s, t)$  between source and target elements as specifications in both forward and reverse directions. Model finding using Alloy is used to bidirectionalise ATL in [12].

In this paper we describe an approach to bx definition using standard UML 2 elements (class diagrams, use cases, OCL) which avoids the use of specialised languages, and therefore is preferable from the viewpoint of industrial uptake, and for the long-term maintenance and reuse of bx. In our approach, unidirectional transformations  $\tau$  are specified using use cases and OCL, then a statically-computed bx relation  $Post_\tau$  &  $Inv_\tau$  for  $\tau$  is derived, together with a forward transformation  $\tau^\rightarrow$  extending  $\tau$ , and an inverse  $\tau^\leftarrow$  of  $\tau^\rightarrow$ . The bx relation and transformations are derived from  $\tau$  using a higher-order transformation. This is generally more efficient than the use of constraint programming and model finding.

## 3 Patterns for Bidirectional Transformations

We have identified a number of patterns for bx, which can be used to establish the bx correctness conditions [11]:

- **Auxiliary Correspondence Model:** maintain a detailed trace between source model and target model elements to facilitate change-propagation in source to target or target to source directions.
- **Cleanup before Construct:** for  $R^\rightarrow$ , remove superfluous target model elements which are not in the transformation relation  $R$  with any source elements, before constructing target elements related to source elements, and similarly for  $R^\leftarrow$ .
- **Unique Instantiation:** Do not recreate elements  $t$  in one model which already correspond to an element  $s$  in the other model, instead modify the

data of  $t$  to enforce the transformation relation. Use key attributes to identify when elements should be created or updated.

All three of these patterns are used in our approach to define bx using UML. In addition, we have used three further patterns to structure bx:

- **Factor a Text-to-text bx by a Model-to-model bx:** this architectural pattern defines a text-to-text bx as a composition of parsing, model-to-model and model-to-text transformations in both directions.
- **Inverse Recursive Descent:** if a forward transformation  $\tau^{\rightarrow}$  is defined using structural recursion on the source language  $SL$ , the reverse transformation  $\tau^{\leftarrow}$  may be derived from  $\tau^{\rightarrow}$  as a dual structural recursion on  $TL$ .
- **Bx Data Correspondence:** a special case of Auxiliary Correspondence Model which defines an auxiliary association in  $SL \cup TL$  to express a bx relation concerning specific elements or basic values (eg., a correspondence between expression operators in one language and the other).

## 4 Bidirectional Transformation Specification in UML-RSDS

UML-RSDS is a particular version of executable UML (xUML), based on UML 2 and OCL 2.4, with a formal semantics [8] and an established toolset [9]. Model transformations  $\tau$  are specified in UML-RSDS as UML use cases, defined declaratively by two main predicates, expressed in a subset of OCL:

1. Postconditions  $Post_{\tau}$  which define the intended effect of the transformation at its termination. These are an ordered conjunction  $C_1 \ \& \ \dots \ \& \ C_n$  of OCL constraints (also termed *rules* in the following) and also serve to define a design (activity) and implementation of the transformation.
2. Invariants  $Inv_{\tau}$  which define expected invariant properties which should hold during the transformation execution. These can be derived from  $Post_{\tau}$ , or can be specified explicitly by the developer. They are the constraints of the use case, considered as a UML classifier.

A UML activity  $stat(Cn)$  is derived by the UML-RSDS tools for each post-condition constraint  $Cn$ . This activity is constructed so that it establishes  $Cn$ :  $[stat(Cn)]Cn$ . The activity for  $Post_{\tau}$  is formed as a composition of the  $stat(Cn)$ . In this paper,  $stat(Post_{\tau})$  is the sequential composition of the  $stat(Cn)$ .

Data-dependency relations between the *Post* constraints are important in ensuring the correctness of both unidirectional and bidirectional UML-RSDS transformations. Let  $rd(Cn)$  denote the read-frame of a constraint  $Cn$ , the set of entity type names and data features that it reads, and  $wr(Cn)$  denote its write frame. These are defined in [8].

A dependency ordering  $Cn < Cm$  is defined between constraints by  $wr(Cn) \cap rd(Cm) \neq \{\}$  “ $Cm$  depends on  $Cn$ ”. A use case with postconditions  $C_1, \dots, C_n$  should satisfy the *syntactic non-interference* conditions:

1. If  $C_i < C_j$ , with  $i \neq j$ , then  $i < j$ .
2. If  $i \neq j$  then  $wr(C_i) \cap wr(C_j) = \{\}$ .

Together, these conditions ensure that the activities  $stat(C_j)$  of subsequent constraints  $C_j$  cannot invalidate earlier constraints  $C_i$ , for  $i < j$ .

A constraint is termed a *type 1* constraint if its write and read frames are disjoint. Such constraints usually have an implementation as a bounded loop, as opposed to a fixed-point/unbounded iteration.

An invariant  $Inv$  and explicit inverse predicate  $Post^\sim$  can be derived mechanically from  $Post$  [9, 11]. This derivation is automated in the UML-RSDS tools. Tables 1 and 2 show some examples of inverses  $P^\sim$  of predicates  $P$ . The transformation developer can also specify inverses for particular  $Cn$  by defining a suitable  $Cn^\sim$  constraint in  $Inv$ , for example, to express that a predicate  $t.z = s.x + s.y$  should be inverted as  $s.x = t.z - s.y$ .

$P(s, t)$	$P^\sim(s, t)$	Condition
$t.g = s.f$	$s.f = t.g$	Assignable features $f, g$
$t.g = s.f \rightarrow pow(x)$	$s.f = t.g \rightarrow pow(1.0/x)$	$x$ non-zero, independent of $s.f$
$t.b2 = not(s.b1)$	$s.b1 = not(t.b2)$	Boolean attributes $b1, b2$
$t.g = K * s.f + L$ Numeric constants $K, L$ $K \neq 0$	$s.f = (t.g - L)/K$	$f, g$ numeric attributes
$R(s, t) \ \& \ Q(s, t)$	$R^\sim(s, t) \ \& \ Q^\sim(s, t)$	
$s \rightarrow forAll(x \mid$ $F \rightarrow exists(y \mid P(x, y) \ \& \ y : t))$	$t \rightarrow forAll(y \mid$ $E \rightarrow exists(x \mid P^\sim(x, y) \ \& \ x : s))$	$E$ is the entity element type of $s$

**Table 1.** Inverse of Predicates

In terms of the framework of [16], the source-target relation  $R_\tau$  associated with a UML-RSDS transformation  $\tau$  is  $Post_\tau \ \& \ Inv_\tau$ .  $R_\tau$  is not necessarily bijective. The forward direction of  $\tau$  is normally computed as  $stat(Post_\tau)$ : the UML activity derived from  $Post_\tau$  when interpreted procedurally [8]. However, in order to achieve the correctness and hippocraticness properties,  $Inv_\tau$  must also be considered: before  $stat(Post_\tau)$  is applied to the source model  $m$  and target model  $n$ ,  $n$  must be cleared of elements which fail to satisfy  $Inv_\tau$ . This is a case of the Cleanup before Construct pattern.

Given a typical postcondition  $C_i$  of the form

$$S_i :: \\ SCond_i(self) \Rightarrow T_j \rightarrow exists(t \mid TCond_j(t) \ \& \ P_i(self, t))$$

the corresponding invariant constraint is  $Inv_i$ :

$$T_j :: \\ TCond_j(self) \Rightarrow S_i \rightarrow exists(s \mid SCond_i(s) \ \& \ P_i(s, self))$$

$P(s, t)$	$P^\sim(s, t)$	Condition
$t.rr = s.r \rightarrow reverse()$	$s.r = t.rr \rightarrow reverse()$	$r, rr$ ordered association ends
$t.rr = s.r \rightarrow first()$	$s.r \rightarrow first() = t.rr$	$r$ ordered association end
$t.rr = s.r \rightarrow last()$	$s.r \rightarrow last() = t.rr$	$r$ ordered association end
$t.rr = s.r \rightarrow including(s.p)$	$s.r = t.rr \rightarrow front() \ \&$	$rr, r$ ordered association ends
$t.rr = s.r \rightarrow append(s.p)$	$s.p = t.rr \rightarrow last()$	$p$ 1-multiplicity end
$t.rr = Sequence\{s.p1, s.p2\}$	$s.p1 = t.rr \rightarrow at(1) \ \&$ $s.p2 = t.rr \rightarrow at(2)$	$rr$ ordered association end $p1, p2$ 1-multiplicity ends
$t.rr = s.r \rightarrow select(P)$	$s.r = (s.r - (s.r \rightarrow select(P) - t.rr)) \rightarrow$ $merge(t.rr \rightarrow select(P))$	$r, rr$ set-valued
$t.rr = s.r \rightarrow sort()$	$s.r = t.rr \rightarrow asSet()$	$r$ set-valued, $rr$ ordered
$t.rr = s.r \rightarrow asSequence()$		

**Table 2.** Inverse of Predicates on Associations

The general cleanup constraint derived from  $Inv_i$  is:

$$T_j :: \\ TCond_j(self) \ \& \\ not(S_i \rightarrow exists(s \mid SCond_i(s) \ \& \ P_i(s, self))) \Rightarrow self \rightarrow isDeleted()$$

However, this constraint may delete more  $T_j$  instances than is necessary: not only instances which correspond (i) to deleted  $S_i$  instances, or (ii) to  $S_i$  instances that no longer satisfy  $SCond_i$ , but also (iii) to  $S_i$  instances satisfying  $SCond_i$ , but not  $P_i(s, self)$ . In this last case,  $self$  should not be deleted, instead  $stat(P_i)$  should be performed to re-establish  $P_i$  between the modified  $s$  and  $self$ . Therefore, some form of tracing is required, to distinguish these cases.

The Auxiliary Correspondence Model pattern can be used to introduce tracing relations between source and target entities. In general such relations can be many-many, and are implemented by an auxiliary association  $S_i \overset{*}{\$rtrace} T_j$  linking corresponding elements in the two languages. In the transformation specification,  $s.\$rtrace$  can be referred to as  $s \rightarrow equivalents()$ , and  $t.\$rtrace$  as  $t \rightarrow equivalents()$ . The computational overhead of maintaining such relations can be substantial, and instead we usually apply the Auxiliary Correspondence Model pattern to introduce String-valued identity attributes (primary keys) for source and target entities to record and identify which source and target instances correspond: source instance  $s$  corresponds to target instance  $t$  when  $s.sId = t.tId$ . This also means that  $S_i[t.tId] = s$  and  $T_j[s.sId] = t$ . Finally, the Unique Instantiation pattern is used to modify existing objects in one model which correspond to a modified object in the other model, instead of creating a new object.

If identity attributes have been introduced using Auxiliary Correspondence Model, the cleanup constraints can be simplified to:

$$T_j :: \\ TCond_j(self) \ \& \\ not(tId : S_i \rightarrow collect(sId)) \Rightarrow self \rightarrow isDeleted()$$

and

$$T_j :: \\ TCond_j(self) \ \& \ tId : S_i \rightarrow collect(sId) \ \& \\ not(SCond_i(S_i[tId])) \Rightarrow self \rightarrow isDeleted()$$

which correspond to cases (i) and (ii) above. This version is also potentially more efficient than the original cleanup constraint. The second case is omitted if  $SCond_i$  is the default *true*. The updates of case (iii) are carried out in the *Post* constraints.  $\tau^\rightarrow$  is defined to have postconditions consisting of the cleanup constraints, followed by *Post*, and hence it is an extension of  $\tau$ : its postconditions imply those of  $\tau$ .

The constraints of  $Post_\tau$  need to be interpreted using Unique Instantiation in  $\tau^\rightarrow$ : the  $E \rightarrow exists(e \mid P)$  quantifier in rule succedents should be interpreted as “create a new  $e : E$  and establish  $P$  for  $e$ , unless there already exists an  $e : E$  satisfying  $P$ ”. This is also known as ‘check before enforce’ semantics. If key attributes are defined, a constraint

$$E :: \\ F \rightarrow exists(f \mid f.fId = eId \ \& \ P)$$

means that if  $eId : F \rightarrow collect(fId)$  (a corresponding  $F$  instance  $F[eId]$  already exists), then it is selected and modified to satisfy  $P$  – this performs case (iii) of the cleanup actions to re-establish *Inv* after a source model change. Only if  $eId \notin F \rightarrow collect(fId)$  is a new  $F$  instance created.

## 5 Correctness and Hippocraticness Properties for $\tau^\rightarrow$

In this section we give a formal justification that bx defined using the UML specification approach described above do satisfy the bx properties.

We consider first the case of UML-RSDS transformations that satisfy the Structure Preservation pattern [10], in which source language entities  $S_i$  correspond to target language entities  $T_i$ , and whose constraints are of type 1, satisfying syntactic non-interference. Identity attributes are used to implement Auxiliary Correspondence Model. This means that  $\tau$  is a separate-models transformation with source language  $S$  and target language  $T$ , and postcondition *Post* is an ordered conjunction of constraints  $C_i$  each of the form:

$$S_i :: \\ SCond_i(self) \Rightarrow \\ T_i \rightarrow exists(t \mid t.tId = sId \ \& \ TCond_i(t) \ \& \ P_i(self, t))$$

and *Inv* is a conjunction of constraints  $Inv_i$  of the form

$$T_i :: \\ TCond_i(self) \Rightarrow \\ S_i \rightarrow exists(s \mid tId = s.sId \ \& \ SCond_i(s) \ \& \ P_i(s, self))$$

The simplified cleanup constraints for  $\tau^{\rightarrow}$  are therefore:

$$T_i :: TCond_i(self) \ \& \ not(tId : S_i \rightarrow collect(sId)) \Rightarrow self \rightarrow isDeleted()$$

and

$$T_i :: TCond_i(self) \ \& \ tId : S_i \rightarrow collect(sId) \ \& \ not(SCond_i(S_i[tId])) \Rightarrow self \rightarrow isDeleted()$$

The second constraint is omitted if  $SCond_i$  is the default *true* predicate. We denote the collection of the cleanup constraints by *Cleanup*.

Bx correctness holds, since the cleanup constraints together with *Post* establish *Inv*: the *Cleanup* constraints remove any target model elements that fail to correspond by identity to source model elements in the domain of  $\tau$ , and *Post* modifies target model elements that do correspond to valid source elements, in order to re-establish *Inv*. By construction,  $stat(Post)$  preserves *Inv* and establishes *Post*. Therefore, the sequential composition of  $stat(Cleanup)$  and  $stat(Post)$  establishes the bx relation *R* as *Post* & *Inv*.

Hippocraticness holds, since  $stat(Cleanup)$  has no effect if the models already satisfy *R*, and neither does  $stat(Post)$ , due to the Unique Instantiation interpretation of the *exists* operator. The principle of least change is satisfied, since changes to the source model either lead to deletion of exactly those target elements (and their incident links) which cannot be modified to correspond to source elements (cases (i) and (ii) above), or to creation/modification of target elements, using  $stat(C_i)$ . But  $stat(C_i)$  is designed as a minimally-intrusive update to the target model which will establish  $C_i$ . Interpreting a quantifier  $T_j \rightarrow exists(t \mid P)$  using Unique Instantiation means that un-necessary creation of target elements is avoided.

A more complex case is Entity Merging, where two or more source entity types  $S_i, S_j$  may map to the same target entity type  $T_i$ . Provided that the respective  $TCond_i$  conditions are pairwise disjoint, the same construction and argument for the bx correctness apply as in the Structure Preservation case. Semantic non-interference of *Post* holds, because the set of  $T_i$  instances produced from  $S_i$  is disjoint from the set produced from  $S_j$ , although syntactic non-interference fails<sup>1</sup>. Entity Splitting involves one source entity type mapping to two or more different target entity types. There are two versions of this pattern: (i) two or more separate postconditions on  $S_i$ , with disjoint  $SCond_i$  conditions, mapping to distinct target entity types; (ii) a single constraint which maps one  $S_i$  instance to multiple linked instances of different  $T_j$ . The first case can be treated in a similar way to the Structure Preservation case, provided that syntactic or semantic non-interference of *Post* holds. The second case is common in

<sup>1</sup> A use case satisfies *semantic non-interference* if for  $i < j$ :  $C_i \Rightarrow [stat(C_j)]C_i$ , where  $[act]P$  is the weakest-precondition of *P* with respect to *act*. Syntactic non-interference implies semantic non-interference, but not conversely.



refinement transformations, and in some migrations. Constraints, eg.,  $C_1$ , will have the form

$$S_1 :: \\ SCond_1(self) \Rightarrow T_1 \rightarrow exists(t1 \mid t1.t1Id = sId \ \& \\ T_2 \rightarrow exists(t2 \mid t2.t2Id = sId \ \& \ TCond_1(t1) \ \& \\ TCond_2(t1, t2) \ \& \ P_1(self, t1, t2)))$$

Both  $t1$  and  $t2$  are given *id* values  $self.sId$ . This should be the only constraint which creates either  $T_1$  or  $T_2$  instances (this restriction is implied by syntactic non-interference).  $Inv_1$  is of the form

$$T_1 :: \\ TCond_1(self) \ \& \ t2 : T_2 \ \& \ t2.t2Id = t1Id \ \& \ TCond_2(self, t2) \Rightarrow \\ S_1 \rightarrow exists(s \mid t1Id = s.sId \ \& \ SCond_1(s) \ \& \ P_1(s, self, t2))$$

The  $t2 : T_2$  expression acts like an additional  $T_2 \rightarrow forAll(t2$  quantifier over the remainder of the constraint.

The cleanup constraints are:

$$T_1 :: \\ TCond_1(self) \ \& \ t2 : T_2 \ \& \ t2.t2Id = t1Id \ \& \ TCond_2(self, t2) \ \& \\ not(S_1 \rightarrow exists(s \mid t1Id = s.sId \ \& \ SCond_1(s))) \Rightarrow \\ self \rightarrow isDeleted() \ \& \ t2 \rightarrow isDeleted()$$

The effect of such a *bx* is that a ‘cluster’ of target instances are related to each source instance, with all elements of the cluster sharing the source instance identity value.

Again,  $stat(Post)$  establishes  $Post$  and preserves  $Inv$ , whilst  $stat(Cleanup)$  together with  $stat(Post)$  establishes  $Inv$ . Thus correctness follows. Hippocraticness holds because target model pairs (or clusters)  $t1, t2, \dots$  are only deleted if they do not correspond to a source instance. Unique Instantiation can apply to multiple *exists* quantifiers in the same manner as to single quantifiers. The principle of least change follows from the minimality of  $stat(C_i)$  as an activity to establish  $C_i$ .

Finally, the Map Objects before Links pattern can be used. This separates the construction of target instances, and the linking of these instances. The linking constraints have the form:

$$S_1 :: \\ T_j[idS_1].rr = TRef[r.idSRef]$$

These define target model association ends  $rr$  from source model association ends  $r$ , looking-up target model elements  $T_j[idS_1]$  and  $TRef[r.idSRef]$  which have already been created by a preceding constraint. However, the linking constraints are not invariants of the transformation, and are omitted from  $Inv$ . The cleanup constraints for such  $\tau$  are defined as for the preceding cases, and by a similar

argument, bx correctness, hippocraticness and least change can be shown. The re-establishment of  $Post$  and  $Inv$  for corresponding source and target elements is now performed by possibly a number of separate  $Post$  constraints.

A similar analysis applies to the reverse transformation  $\tau^{\leftarrow}$ . This has postcondition  $Post_{\tau}^{\sim}$ , the explicit form of  $Inv_{\tau}$ .  $Post_{\tau}^{\sim}$  may involve *view updates* [2] of the source language SL, based on changes to TL elements. For example, predicates  $s.f \rightarrow last() = t.g$  or  $s.f \rightarrow select(P1) = t.g$ .

$stat(Post^{\sim})$  establishes  $Inv$ , whilst the cleanup constraints together with  $Post^{\sim}$  establish  $Post$ . Thus  $\tau^{\leftarrow}$  satisfies correctness. Hippocraticness follows since neither the cleanup or  $Post^{\sim}$  constraints modify the source model if  $R$  already holds.

## 6 Evaluation

The approach described in this paper enables transformation developers to focus on writing correct postconditions  $Post_{\tau}$  for a conventional (unidirectional) forward transformation  $\tau$ . From these, the necessary invariants and cleanup constraints for bx execution mode can be automatically generated by a higher-order transformation, producing a bx relation  $Post_{\tau} \ \& \ Inv_{\tau}$  and forward and reverse transformations with respect to this relation. The forward transformation extends  $\tau$ . This is the same approach as taken for the bidirectionalisation of ATL in [12]: from a unidirectional forward ATL transformation  $\tau$ , a bx relation  $T$  and transformations  $T^{\rightarrow}$  and  $T^{\leftarrow}$  are derived, where  $T^{\rightarrow}$  extends  $\tau$ . In contrast to [12] we use the minimality of the activity  $stat(P)$  to achieve the principle of least change, rather than reliance upon an external formalism (Alloy). Meta-model constraints are incorporated into the bx relation by [12]. Our approach is instead to prove separately that  $Post_{\tau} \ \& \ Inv_{\tau} \Rightarrow Th_{TL}$ , where  $Th_{TL}$  are the target language constraints.

By using standard UML notations and concepts, our approach should enable bx specifications to be retained and maintained independently of particular transformation technologies. The use of OCL for transformation specification facilitates the automated derivation of transformation invariants and reverse rules. Our approach scales up to transformations of practical size, and we have applied it to a complex UML to C code generator with over 150 mapping rules. The approach defined here could also potentially be used with other similar specification formalisms, such as Alf/fUML [15] and EVL (<http://www.eclipse.org/epsilon/doc/evl>).

## References

1. A. Anjorin, G. Varro, A. Schurr, *Complex attribute manipulation in TGGs with constraint-based programming techniques*, BX 2012, Electronic Communications of the EASST vol. 49, 2012.
2. F. Bancilhon, N. Spyrtos, *Update semantics of relational views*, ACM Transactions on Database Systems, vol. 6, no. 4, pp. 557–575, 1981.

3. M. Beine, N. Hames, J. Weber, A. Cleve, *Bidirectional transformations in database evolution: a case study ‘at scale’*, EDBT/ICDT 2014, CEUR-WS.org, 2014.
4. J. Cheney, J. McKinna, P. Stevens, J. Gibbons, *Towards a repository of bx examples*, EDBT/ICDT 2014, 2014, pp. 87–91.
5. A. Cicchetti, D. Di Ruscio, R. Eramo, A. Pierantonio, *JTL: a bidirectional and change propagating transformation language*, SLE 2010, LNCS vol. 6563, 2011, pp. 183–202.
6. F. Jouault, F. Allilaire, J. Bezivin, I. Kurtev, *ATL: a model transformation tool*, Science of Computer Programming, vol. 72, no. 1, pp. 31–39, 2008.
7. D. S. Kolovos and R. F. Paige and F. Polack, *The Epsilon Transformation Language*, ICMT, 2008, pp. 46–60.
8. K. Lano, S. Kolahdouz-Rahimi, *Constraint-based specification of model transformations*, Journal of Systems and Software, vol. 88, no. 2, February 2013, pp. 412–436.
9. K. Lano, *The UML-RSDS Manual*, [www.dcs.kcl.ac.uk/staff/kcl/uml2web/umlrlds.pdf](http://www.dcs.kcl.ac.uk/staff/kcl/uml2web/umlrlds.pdf), 2016.
10. K. Lano, S. Kolahdouz-Rahimi, *Model-transformation Design Patterns*, IEEE Transactions in Software Engineering, vol 40, 2014.
11. K. Lano, S. Kolahdouz-Rahimi, S. Yassipour-Tehrani, *Patterns for Specifying Bidirectional Transformations in UML-RSDS*, ICSEA 2015.
12. N. Macedo, A. Cunha, *Least-change bidirectional model transformation with QVT-R and ATL*, Softw. Syst. Model, 2014, doi:10.1007/s10270-014-0437-x.
13. L. Meertens, *Designing constraint maintainers for user interaction*, Third Workshop on Programmable Structured Documents, Tokyo University, 2005.
14. OMG, *MOF 2.0 Query/View/Transformation Specification v1.1*, 2011.
15. OMG, *Action Language for Foundational UML (ALF)*, v1.0.1, 2015.
16. P. Stevens, *Bidirectional model transformations in QVT: semantic issues and open questions*, SoSyM, vol. 9, no. 1, January 2010, pp. 7–20.
17. P. Stevens, *A simple game-theoretic approach to checkonly QVT-Relations*, Softw. Syst. Model vol. 12, no. 1, 2013, pp. 175–199.